High Performance Computing - HW 4

Anthony Maylath

April 15, 2019

The Courant Institute for Mathematical Sciences, New York University

1 Question 1

I will use module cuda-9.0 to run and compile my code. The user may need to load the cuda-9.0 module before attempting to compile. For the CPU inner product, I simply perform a reduction on a loop that iteratively adds a[i] + b[i] to a sum. I use -O3 optimized openmp for the CPU portion. For the CUDA implementation, I modify function reduction_kernal2() to produce function innerprod_kernel2(). There is exactly one line of code different in these two functions. Instead of initializing smem to $a[idx]$, I initalize to $a[idx] * b[idx]$. Hence, if we call reduction_kernal2() recursively after a call to innerprod_kernel2(), the result will be the inner product of a and b. The full logic is implemented in the function, innerprod_CUDA(). I use this function to compute the inner product of x_d and y_d.

For the matrix multiplication implimentation, I initialize a vector, A which is a matrix of size $N \times N$ in row major order. For the purpose of this question, I intialize $A_{ij} = j$ and compute the matrix vector product Ay by calling innerprod_CUDA() N times for each row of A.

Table 1 through 3 shows the bandwidth on various cuda resources for inner product and matrix multiplication. With "Same Row" matrix vector multiplication, I assume the rows of the matrix are identical and repeat the computation with the same row in memory. WIth the "Diff Row" version, I copy a row onto device memory for every inner product to compute the

resulting vector.

I run my code on cuda1, cuda2, and cuda5. I ran each algorithm with dimension 1,000, 10,000, and 60,000. For every algorithm, the bandwidth increased with dimension size. The performance was a bit worse on cuda5 which makes sense as the resource has fewer cores than cuda1 or cuda2.

Resource	Algorithm	N	Bandwidth (GB/s)
cuda1	Inner Product	1000	0.38
cuda1	Matrix Vector Mult (Same Row)	1000	1.072
cuda1	Matrix Vector Mult (Diff Row)	1000	1.2967
cuda1	Inner Product	10000	3.2168
cuda1	Matrix Vector Mult (Same Row)	10000	8.668
cuda1	Matrix Vector Mult (Diff Row)	10000	8.576
cuda1	Inner Product	60000	36.085
cuda1	Matrix Vector Mult (Same Row)	60000	48.375
cuda1	Matrix Vector Mult (Diff Row)	60000	15.819

Table 1: Performance on cuda1

Resource	Algorithm	N	Bandwidth (GB/s)
cuda2	Inner Product	1000	0.3843
cuda2	Matrix Vector Mult (Same Row)	1000	0.79749
cuda2	Matrix Vector Mult (Diff Row)	1000	0.8831
cuda2	Inner Product	10000	4.036868
cuda2	Matrix Vector Mult (Same Row)	10000	10.19127
cuda2	Matrix Vector Mult (Diff Row)	10000	6.915
cuda2	Inner Product	60000	36.085
cuda2	Matrix Vector Mult (Same Row)	60000	48.375
cuda2	Matrix Vector Mult (Diff Row)	60000	15.819

Table 2: Performance on cuda2

Resource	Algorithm	N	Bandwidth (GB/s)
cuda5	Inner Product	1000	0.38
cuda5	Matrix Vector Mult (Same Row)	1000	0.893
cuda5	Matrix Vector Mult (Diff Row)	1000	1.072
cuda5	Inner Product	10000	3.941
cuda5	Matrix Vector Mult (Same Row)	10000	7.086
cuda5	Matrix Vector Mult (Diff Row)	10000	4.458
cuda5	Inner Product	60000	26.773
cuda5	Matrix Vector Mult (Same Row)	60000	35.401
cuda5	Matrix Vector Mult (Diff Row)	60000	6.734

Table 3: Performance on cuda5

2 Question 2

Note that I use block size = 32 for all code in Question 2. I implement a function, jacobi_kernel(), in file 2dJacobi.cu which computes the Jacobi step. The function uses shared memory for the right hand side, f. I filter out the perimeter of the matrix (first/last row or column) by returning from the function if the index is on the perimeter. The function uses row major matrix implementation with the update step as follows:

```
result[idx] = (f_h + u[idx - N] + u[idx - 1] + u[idx + N] +
 ↪  u[idx + 1])/4;
```

where u is the vector containing the previous jacobi step.

2.1 Extra Credit

I implement two CUDA kernels to facilitate the red/black Gauss-Seidel method. The key step of each of these methods is to identify black and red points respectively. Once we identify these points, they are skipped for the next step of the iteration. For instance, in the kernel, cudaRed(), I skip the black points as follows:

```
bool black = (idx/N % 2 == 0) && (idx % 2 == 1); //If true
 ↪  then black node
black = black || (idx/N % 2 == 1) && (idx % 2 == 0);
```

```
if(black){return;} //If on a black node
```

The logic is similar for function, cudaBlack(). The remainder of the implementation for Gauss-Seidel is similar to my implementation in Homework 2.

When you call an executable version of my code, you can either specify zero or four arguments. If specified, the four arguments are matrix dimension, number of iterations, number of omp threads, and solver (either "GS" or "jacobi"). By default, I have a matrix of size 4 with 4 omp threads, 100 iterations, and the jacobi solver.

2.2 Results

Tables 4 through 6 show the performance on cuda{1,2,5}. cuda2 seems to do better for larger problem sizes while cuda1 performs best on smaller problem sizes. The performance between Jacobi and Gauss-Seidel is roughly the same. However, Gauss-Seidel seems to be a bit slower on average.

Resource	Algorithm	N	Iterations	Time	Bandwidth (GB/s)
cuda1	Jacobi	100	100	0.00079	49.342
cuda1	Jacobi	100	1000	0.0062	64.058
cuda1	Jacobi	1000	1000	0.1892	29.8085
cuda1	Jacobi	1500	100	0.0445	201.962080
cuda1	GS	100	100	0.00078	49.688
cuda1	GS	100	1000	0.0067	59.4022
cuda1	GS	1000	1000	0.1879	30.016
cuda1	GS	1500	100	0.0447	201.3799

Table 4: Performance on cuda1: Laplace

Resource	Algorithm	N	Iterations	Time	Bandwidth (GB/s)
cuda2	Jacobi	100	100	0.00116	33.1825
cuda2	Jacobi	100	1000	0.01024	38.87
cuda2	Jacobi	1000	1000	0.0464	121.4557
cuda2	Jacobi	1500	100	0.012324	726.869
cuda2	GS	100	100	0.001114	34.3567
cuda2	GS	100	1000	0.0122	32.616
cuda2	GS	1000	1000	0.04645	121.3076
cuda2	GS	1500	100	0.012345	725.7375

Table 5: Performance on cuda2: Laplace

Resource	Algorithm	N	Iterations	Time	Bandwidth (GB/s)
cuda5	Jacobi	100	100	0.00114	34.122
cuda5	Jacobi	100	1000	0.0089	44.731
cuda5	Jacobi	1000	1000	0.2219	25.4098
cuda5	Jacobi	1500	100	0.06052	148.6
cuda5	GS	100	100	0.0012	36.5328
cuda5	GS	100	1000	0.0086	46.165
cuda5	GS	1000	1000	0.2352	23.95034
cuda5	GS	1500	100	0.0558	161.152

Table 6: Performance on cuda5: Laplace

3 Question 3

My team will include John Donaghy and Anthony Maylath. We plan to explore MPI implementations of clustering algorithms. We will begin by implementing a serial version of the k-Means algorithm as a baseline. We will then look for ways to engineer speedup using MPI. Time permitting, we will extend the implementation to the EM clustering algorithm.

We will mainly apply our clustering algorithm to the Zillow's Home Value Prediction (Zestimate) dataset found on Kaggle. The dataset contains individual property features from homes in Southern Californa from 2016-2017. Sample fields include Year Built, Overall Condition, and Garage Area. There are a total of 58 features in the dataset. We will initially run our code on a subset of these features (roughly 8) and add more features as time permits.

- **Zillow Dataset:** `https://www.kaggle.com/c/zillow-prize-1/data`